

# *Data structures in R*

## *The base structures*

R.W. Oldford

## Preliminaries to find data (and images for these slides)

```
# A little function that just concatenates paths (as strings)  
# to produce a "path" to some file/directory  
path_concat <- function(path1,  
                          path2,  
                          sep="/") {  
  paste(path1, path2, sep = sep)  
}  
  
# Note that you might have to give a different value  
# for the directory separator (e.g. sep = "\" on Windows?)  
#  
# Here's where my course files are on my machine  
coursesDirectory <- "/Users/rwoldford/Documents/Admin/courses/"  
#  
# Use path_concat() to produce new paths to sub-directories.  
# For example:  
EDA <- path_concat(coursesDirectory, "STAT\ 847")  
dataDirectory <- path_concat(EDA, "data")  
imageDirectory <- path_concat(EDA, "img")
```

## *Data structures in R*

The base data structures in R:

dimensionality	homogeneous contents	heterogeneous contents
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Note there are no scalar or 0-dimensional data structures. Instead these are 1d data structures having a single element.

There are also three different types of object-oriented programming systems in R (**S3**, **S4**, and **reference classes**) which can be used to construct more complex data types.

The function `str()` can be used to reveal the contents of any R data structure.

## *Data structures in R – Vectors*

The basic data structure is a “vector”

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds:

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:



## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

- ▶ its type, `typeof()`

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

- ▶ its type, `typeof()`
- ▶ the number of elements it has, `length()`

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

- ▶ its type, `typeof()`
- ▶ the number of elements it has, `length()`
- ▶ a place for arbitrary additional properties, `attributes()`

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

- ▶ its type, `typeof()`
- ▶ the number of elements it has, `length()`
- ▶ a place for arbitrary additional properties, `attributes()`

Elements of an **atomic vector** must all be of the **same** type.

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

- ▶ its type, `typeof()`
- ▶ the number of elements it has, `length()`
- ▶ a place for arbitrary additional properties, `attributes()`

Elements of an **atomic vector** must all be of the **same** type.

Elements of a **list** can be of different types.

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

- ▶ its type, `typeof()`
- ▶ the number of elements it has, `length()`
- ▶ a place for arbitrary additional properties, `attributes()`

Elements of an **atomic vector** must all be of the **same** type.

Elements of a **list** can be of different types.

Constructors: `c()` for atomic vectors, `list()` for lists.

## *Data structures in R – Vectors*

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

- ▶ its type, `typeof()`
- ▶ the number of elements it has, `length()`
- ▶ a place for arbitrary additional properties, `attributes()`

Elements of an **atomic vector** must all be of the **same** type.

Elements of a **list** can be of different types.

Constructors: `c()` for atomic vectors, `list()` for lists.

tests: `is.atomic()` and `is.list()`.

## Data structures in R – `c()` constructing atomic vectors

Atomic vectors are constructed using `c()` (c for “combine”)

```
x <- c(1, 2, 3)
x
```

```
## [1] 1 2 3
```

```
is.atomic(x)
```

```
## [1] TRUE
```

```
is.list(x)
```

```
## [1] FALSE
```

Atomic vectors are always “flat”

```
y <- c(x, x, 4, 5, 6)
y
```

```
## [1] 1 2 3 1 2 3 4 5 6
```

```
c(y, c(7, 8, x, 9, c(10, 11)))
```

```
## [1] 1 2 3 1 2 3 4 5 6 7 8 1 2 3 9 10 11
```



## *Data structures in R – c() constructing atomic vectors*

Elements of an atomic vector are accessed using the `[]` operator (see `?[]`)

```
x <- c("a", "b", "c", "d", "e", "f")  
x[3]
```

```
## [1] "c"
```

```
x[c(1,3,5)]
```

```
## [1] "a" "c" "e"
```

And set with the same function

```
x[1] <- "EH"  
x
```

```
## [1] "EH" "b"  "c"  "d"  "e"  "f"
```

```
x[c(3,5)] <- c("third", "fifth")  
x
```

```
## [1] "EH"      "b"      "third" "d"      "fifth" "f"
```

## *Data structures in R – double precision numeric vectors*

```
x <- c(1, 2, 3)
length(x)
```

```
## [1] 3
```

```
typeof(x)
```

```
## [1] "double"
```

```
attributes(x)
```

```
## NULL
```

```
is.atomic(x)
```

```
## [1] TRUE
```

```
is.numeric(x)
```

```
## [1] TRUE
```

```
is.double(x)
```

```
## [1] TRUE
```

## Data structures in R – integer numeric vectors

```
x <- c(1L, 20L, 3L) # "longs"  
length(x)
```

```
## [1] 3
```

```
typeof(x)
```

```
## [1] "integer"
```

```
attributes(x)
```

```
## NULL
```

```
is.atomic(x)
```

```
## [1] TRUE
```

```
is.numeric(x)
```

```
## [1] TRUE
```

```
is.integer(x)
```

```
## [1] TRUE
```

## *Data structures in R – logical vectors*

```
x <- c(T, F, TRUE, T, FALSE, T)
length(x)
```

```
## [1] 6
```

```
typeof(x)
```

```
## [1] "logical"
```

```
attributes(x)
```

```
## NULL
```

```
is.atomic(x)
```

```
## [1] TRUE
```

```
is.numeric(x)
```

```
## [1] FALSE
```

```
is.logical(x)
```

```
## [1] TRUE
```

## *Data structures in R – character vectors*

```
x <- c("Now", "is the time", "for", "all")  
length(x)
```

```
## [1] 4
```

```
typeof(x)
```

```
## [1] "character"
```

```
attributes(x)
```

```
## NULL
```

```
is.atomic(x)
```

```
## [1] TRUE
```

```
is.numeric(x)
```

```
## [1] FALSE
```

```
is.character(x)
```

```
## [1] TRUE
```

## *Data structures in R – type contagion*

From least to most flexible vector types are: logical, integer, double, and character.

Elements are coerced to be of the **same** type (the most flexible).

```
typeof(c(FALSE, T))
```

```
## [1] "logical"
```

```
typeof(c(FALSE, T, 2L))
```

```
## [1] "integer"
```

```
typeof(c(FALSE, T, 2L, 3))
```

```
## [1] "double"
```

```
typeof(c(FALSE, T, 2L, 3, "four"))
```

```
## [1] "character"
```

```
c(FALSE, T, 2L, 3, "four")
```

```
## [1] "FALSE" "TRUE"  "2"      "3"      "four"
```

All elements are automatically coerced to be strings.

## Data structures in R – coercion

Can force the coercion using `as.numeric()`, `as.double()`, `as.integer()`, or `as.logical()`

```
as.numeric(c(FALSE, T, TRUE, F, F))
```

```
## [1] 0 1 1 0 0
```

```
as.double(c(FALSE, T, TRUE, F, F))
```

```
## [1] 0 1 1 0 0
```

```
as.integer(c(FALSE, T, TRUE, F, F))
```

```
## [1] 0 1 1 0 0
```

```
as.character(c(FALSE, T, TRUE, F, F))
```

```
## [1] "FALSE" "TRUE"  "TRUE"  "FALSE" "FALSE"
```

```
as.logical(c(0, 1, 2.3, 4.5, 6))
```

```
## [1] FALSE TRUE TRUE TRUE TRUE
```

Note that many functions will force their argument to the required type.

E.g. `sum()` forces its argument to be numeric, logical operators `&`, `|`, etc. force theirs to be logical.

## *Data structures in R – coercion*

Forcing coercion can result in the loss of information and can give some strange answers:

```
as.numeric(c(FALSE, T, 2L, 3))
```

```
## [1] 0 1 2 3
```

```
as.numeric(c(FALSE, T, 2L, 3, "four"))
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA 2 3 NA
```

```
as.numeric(c(as.numeric(c(FALSE, T, 2L, 3)), "four"))
```

```
## Warning: NAs introduced by coercion
```

```
## [1] 0 1 2 3 NA
```

Note that warnings are given.



## *Data structures in R – vectors*

Can also produce a vector (possibly to be modified later) by specifying its type (mode) and length:

```
x <- vector(mode = "double", length = 3)
x
```

```
## [1] 0 0 0
```

```
y <- vector(mode = "logical", length = 3)
y
```

```
## [1] FALSE FALSE FALSE
```

```
z <- vector(mode = "character", length = 3)
z
```

```
## [1] "" "" ""
```

And

## Data structures in R – Lists

Elements of lists can be of any type:

```
x <- list("a", c(2, 3, 4), c(T,F), c("b", "c", "d", 56))
x
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 2 3 4
##
## [[3]]
## [1] TRUE FALSE
##
## [[4]]
## [1] "b" "c" "d" "56"
```

```
str(x)
```

```
## List of 4
## $ : chr "a"
## $ : num [1:3] 2 3 4
## $ : logi [1:2] TRUE FALSE
## $ : chr [1:4] "b" "c" "d" "56"
```

```
attributes(x)
```

```
## NULL
```

Note the **double** square brackets now appearing!

## Data structures in R – Lists

Elements of lists can be accessed in a few ways:

```
x[2]
```

```
## [[1]]  
## [1] 2 3 4
```

```
typeof(x[2])
```

```
## [1] "list"
```

```
length(x[2])
```

```
## [1] 1
```

```
x[[2]]
```

```
## [1] 2 3 4
```

```
typeof(x[[2]])
```

```
## [1] "double"
```

And can be created using `vector()`, default elements being NULL (being an “empty” vector)

```
vector(mode = "list", length = 3)[[1]]
```

```
## NULL
```

## *Data structures in R – Lists are recursive*

```
x <- list("one", 2)
is.recursive(x)
```

```
## [1] TRUE
```

Lists can be arbitrarily recursive:

```
y <- list(x, list(3, x))
is.recursive(y)
```

```
## [1] TRUE
```

```
str(y)
```

```
## List of 2
## $ :List of 2
## ..$ : chr "one"
## ..$ : num 2
## $ :List of 2
## ..$ : num 3
## ..$ :List of 2
## .. ..$ : chr "one"
## .. ..$ : num 2
```

## *Data structures in R – Vectors can have named components*

### Atomic vectors

```
x <- c(a=3, b=4, 5)
x
```

```
## a b
## 3 4 5
```

```
names(x)
```

```
## [1] "a" "b" ""
```

```
x["a"]
```

```
## a
## 3
```

```
attributes(x)
```

```
## $names
## [1] "a" "b" ""
```

## *Data structures in R – Vectors can have named components*

Lists (or “recursive vectors”)

```
x <- list(a=c(1,2,3), b=4, 5)
str(x)
```

```
## List of 3
## $ a: num [1:3] 1 2 3
## $ b: num 4
## $ : num 5
```

```
x["b"]
```

```
## $b
## [1] 4
```

```
x$b
```

```
## [1] 4
```

```
names(x)
```

```
## [1] "a" "b" ""
```

## Data structures in R – Lists as general data structures

Lists (handy with named components) can be used to create more general structures. For example,

```
# data frames:  
is.list(cars)
```

```
## [1] TRUE
```

```
names(cars)
```

```
## [1] "speed" "dist"
```

```
# results of functions  
fit <- lm(dist ~ speed, data = cars)  
is.list(fit)
```

```
## [1] TRUE
```

```
names(fit)
```

```
## [1] "coefficients" "residuals"      "effects"      "rank"  
## [5] "fitted.values" "assign"         "qr"          "df.residual"  
## [9] "xlevels"      "call"          "terms"       "model"
```

## Data structures in R – Lists to atomic vectors

Sometimes it can be handy to change a list to an atomic vector:

```
x <- list(a = 1, b = c(2, 3, 4), c = list(e = c(5, 6), f = c(7, 8, 9)))  
str(x)
```

```
## List of 3  
## $ a: num 1  
## $ b: num [1:3] 2 3 4  
## $ c:List of 2  
## ..$ e: num [1:2] 5 6  
## ..$ f: num [1:3] 7 8 9
```

```
y <- unlist(x)  
is.list(y)
```

```
## [1] FALSE
```

```
y
```

```
##      a      b1      b2      b3 c.e1 c.e2 c.f1 c.f2 c.f3  
##      1      2      3      4      5      6      7      8      9
```

```
names(y)
```

```
## [1] "a"      "b1"     "b2"     "b3"     "c.e1"   "c.e2"   "c.f1"   "c.f2"   "c.f3"
```



## Data structures in R – attributes

Additional information can be added to any R object as one or more attributes.

- ▶ these are very much like a property list (or plist) in other languages.

```
y
```

```
##      a    b1    b2    b3 c.e1 c.e2 c.f1 c.f2 c.f3
##      1     2     3     4     5     6     7     8     9
```

```
names(y)
```

```
## [1] "a"      "b1"     "b2"     "b3"     "c.e1"   "c.e2"   "c.f1"   "c.f2"   "c.f3"
```

```
# introduce a specific attribute
attr(y, "originalNames") <- c("a", "b", "c", "d", "e", "f")
attr(y, "originalNames")
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

which provides a simple cache of the component names from the original list hierarchy.

## Data structures in R – attributes

Using `attr()` to set a specific named attribute on the object modifies that object:

```
tau <- 2 * pi
attr(tau, "description") <- "Twice pi"
attributes(tau)
```

```
## $description
## [1] "Twice pi"
```

`structure()` is similar but returns a copy:

```
TwoPi <- structure(tau, description = "Two times pi")
attributes(TwoPi)
```

```
## $description
## [1] "Two times pi"
```

```
# TwoPi is a different object having the same value as tau
# which can be checked here since it is just a numeric vector
tau == TwoPi
```

```
## [1] TRUE
```

## *Data structures in R – attributes*

Of course, assigning a value to `attributes()` will change all attributes on that object.

```
attributes(tau)
```

```
## $description  
## [1] "Twice pi"
```

```
attributes(tau) <- list(constant = "tau")
```

and we have lost the previous attributes.

```
attributes(tau)
```

```
## $constant  
## [1] "tau"
```

## Data structures in R – factors

A factor is a special kind of atomic vector having class factor and an attribute called levels. A factor is typically used to represent a categorical variate having a fixed, finite, and known set of values. The possible values are assigned to the factor levels.

```
treatment <- factor(c("a", "b", "b", "b", "a", "c", "c", "a", "b"))
levels(treatment)
```

```
## [1] "a" "b" "c"
```

```
table(treatment)
```

```
## treatment
## a b c
## 3 4 2
```

```
# levels are fixed
treatment[3] <- "d"
```

```
## Warning in `[<-.factor`(`*tmp*`, 3, value = "d"): invalid factor level, NA
## generated
```

```
treatment
```

```
## [1] a    b    <NA> b    a    c    c    a    b
## Levels: a b c
```

## *Data structures in R – factors*

A factor is an atomic vector,

```
is.atomic(treatment)
```

```
## [1] TRUE
```

```
typeof(treatment)
```

```
## [1] "integer"
```

of type integer having a length and attributes

```
length(treatment)
```

```
## [1] 9
```

```
attributes(treatment)
```

```
## $levels  
## [1] "a" "b" "c"  
##  
## $class  
## [1] "factor"
```

## *Data structures in R – factors*

These attributes can be accessed via two functions

```
levels(treatment)
```

```
## [1] "a" "b" "c"
```

```
class(treatment)
```

```
## [1] "factor"
```

## *Data structures in R – factors*

All factor levels need to be known, but need not appear:

```
answer <- factor(c(1, 2, 1, 1, 2, 2, 1, 1, 2), levels = c(1, 2, 3))  
table(answer)
```

```
## answer  
## 1 2 3  
## 5 4 0
```

Again, answer is a factor and not

```
is.integer(answer)
```

```
## [1] FALSE
```

## Data structures in R – factors

Though a factor is an atomic vector of type integer, it does not test positive as an integer

```
is.integer(treatment)
```

```
## [1] FALSE
```

Nor does it test positive as a vector

```
is.vector(treatment)
```

```
## [1] FALSE
```

The problem is that `is.vector()` returns TRUE only if its argument is a vector **and** it has no attributes except (possibly) a names attribute.

```
names(treatment)
```

```
## NULL
```

It is a factor

```
is.factor(treatment)
```

```
## [1] TRUE
```



## Data structures in R – factors

A factor could be coerced to a vector

```
as.vector(treatment)
```

```
## [1] "a" "b" NA  "b" "a" "c" "c" "a" "b"
```

```
typeof(treatment)
```

```
## [1] "integer"
```

but loses the levels information. Combining factors will perform a similar coercion:

```
c(treatment)
```

```
## [1]  1  2 NA  2  1  3  3  1  2
```

```
is.integer(c(treatment))
```

```
## [1] TRUE
```

## Data structures in R – accidental factors

Often, we read in data from, say, a .csv file where we might be expecting only numerical values. For example, the csv file might have contents like this:

```
x, y  
5, 3  
7, -  
8, 2
```

The dash " – " is a mistake in coding the data, or perhaps codes missing data. Read in (using `read.csv()`), the result assigned to `data`:

```
# for example  
data <- read.csv(path_concat(dataDirectory, "fake.csv"), header = TRUE, sep=",")
```

the dash – will be located in the same place in data.

```
data
```

```
##    x y  
## 1 5 3  
## 2 7 -  
## 3 8 2
```

Where we might have expected a numeric atomic vector, `data$y` is a factor (Why?)

```
data$y
```

```
## [1] 3 - 2  
## Levels: - 2 3
```

## Data structures in R – matrices and arrays

An atomic vector can be made to act like a matrix (or array) by simply adding an attribute `dim` to record the matrix (or array) dimensions.

```
x <- 1:12
dim(x) <- c(2,6)
x
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    7    9   11
## [2,]    2    4    6    8   10   12
```

```
class(x)
```

```
## [1] "matrix"
```

```
attributes(x)
```

```
## $dim
## [1] 2 6
```

```
typeof(x)
```

```
## [1] "integer"
```

```
length(x)
```

```
## [1] 12
```

## Data structures in R – matrices and arrays

An atomic vector can be made to act like a matrix (or array) by simply adding a attribute `dim` to record the matrix (or array) dimensions.

```
x <- 1:12
dim(x) <- c(2,3,2)
x
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

```
class(x)
```

```
## [1] "array"
```

```
attributes(x)
```

```
## $dim
## [1] 2 3 2
```

## Data structures in R – matrices and arrays

There are also special constructor functions

```
# for matrices  
x <- matrix(1:4, nrow = 3, ncol = 4)  
x
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    1    4    3    2  
## [2,]    2    1    4    3  
## [3,]    3    2    1    4
```

```
dim(x)
```

```
## [1] 3 4
```

```
c(nrow(x), ncol(x))
```

```
## [1] 3 4
```

```
length(x)
```

```
## [1] 12
```

## Data structures in R – matrices and arrays

There are also special constructor functions

```
# For arrays  
y <- array(1:8, dim = c(2,3,2))  
y
```

```
## , , 1  
##  
##      [,1] [,2] [,3]  
## [1,]    1    3    5  
## [2,]    2    4    6  
##  
## , , 2  
##  
##      [,1] [,2] [,3]  
## [1,]    7    1    3  
## [2,]    8    2    4
```

```
dim(y)
```

```
## [1] 2 3 2
```

```
c(nrow(y), ncol(y))
```

```
## [1] 2 3
```

```
length(y)
```

```
## [1] 12
```

## Data structures in R – matrices and arrays

Again, these are atomic vectors

```
y <- array(1:6, dim = c(2,3,2))  
is.atomic(y)
```

```
## [1] TRUE
```

```
length(y)
```

```
## [1] 12
```

```
typeof(y)
```

```
## [1] "integer"
```

```
attributes(y)
```

```
## $dim  
## [1] 2 3 2
```

```
is.vector(y) # explain this result
```

```
## [1] FALSE
```

# Data structures in R – matrices and arrays

## Subsetting

```
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    3    2
## [2,]    2    1    4    3
## [3,]    3    2    1    4
```

```
x[c(1,2), c(2,4)]
```

```
##      [,1] [,2]
## [1,]    4    2
## [2,]    1    3
```

```
x[c(T,F,T), c(F,T,T,F)]
```

```
##      [,1] [,2]
## [1,]    4    3
## [2,]    2    1
```

```
x[-2,]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    3    2
## [2,]    3    2    1    4
```

```
x[2,]
```

```
## [1] 2 1 4 3
```



## Data structures in R – matrices and arrays

Dimensions are dropped by default:

```
x
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    1    4    3    2  
## [2,]    2    1    4    3  
## [3,]    3    2    1    4
```

```
x[2,]
```

```
## [1] 2 1 4 3
```

```
dim(x[2,])
```

```
## NULL
```

```
x[2,, drop = FALSE] # preserve the dimensions
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    2    1    4    3
```

```
dim(x[2,, drop = FALSE])
```

```
## [1] 1 4
```

## Data structures in R – matrices and arrays

Matrices/arrays are still indexable as vectors:

```
x
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    1    4    3    2  
## [2,]    2    1    4    3  
## [3,]    3    2    1    4
```

```
x[7]
```

```
## [1] 3
```

```
x[-7]
```

```
## [1] 1 2 3 4 1 2 4 1 2 3 4
```

```
x[[5]]
```

```
## [1] 1
```

See `help("[")` for details.

## Data structures in R – matrices and arrays

Subsetting is same on arrays

```
y
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
y[1,2,1, drop=FALSE]
```

```
## , , 1
##
##      [,1]
## [1,]    3
```

```
y[1,2,]
```

```
## [1] 3 3
```

```
y[, -2, 1]
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
```

## Data structures in R – matrices and arrays

The transpose `t()` switches the rows and columns of a matrix. That is it permutes the indices:

```
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    3    2
## [2,]    2    1    4    3
## [3,]    3    2    1    4
```

```
t(x)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    1    2
## [3,]    3    4    1
## [4,]    2    3    4
```

## Data structures in R – matrices and arrays

For arrays, `aperm()` can be used to permute indices:

```
y
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
aperm(y, perm = c(2,1,3))
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
```

## Data structures in R – matrices and arrays

Just as vectors are combined with `c()`, conformable matrices can be combined using `cbind()` and `rbind()` (column and row binding).

```
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    3    2
## [2,]    2    1    4    3
## [3,]    3    2    1    4
```

```
cbind(x, 11:13, x[,3:4])
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    4    3    2   11    3    2
## [2,]    2    1    4    3   12    4    3
## [3,]    3    2    1    4   13    1    4
```

```
rbind(x, 11:14)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    3    2
## [2,]    2    1    4    3
## [3,]    3    2    1    4
## [4,]   11   12   13   14
```

## Data structures in R – matrices and arrays

An exception to conformable matrices when using `cbind()` and `rbind()` is the case when one of the arguments is a scalar.

```
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    3    2
## [2,]    2    1    4    3
## [3,]    3    2    1    4
```

```
cbind(x, 1000, x)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    4    3    2 1000    1    4    3    2
## [2,]    2    1    4    3 1000    2    1    4    3
## [3,]    3    2    1    4 1000    3    2    1    4
```

## Data structures in R – matrices and arrays

Similarly for arrays, the combine constructor `abind()` (from the package `abind`) is available to extend an array along different dimensions

```
library(abind)
abind(x, 11:13, x[,3:4], along = 2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    4    3    2   11    3    2
## [2,]    2    1    4    3   12    4    3
## [3,]    3    2    1    4   13    1    4
```

```
abind(y, 100 * x[c(2,3),c(3:4)], along = 2)
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5  400
## [2,]    2    4    6  100
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5  300
## [2,]    2    4    6  400
```

Note that the default `along` is the last dimension and `along = 0` will create a new dimension at the front.



## Data structures in R – matrices and arrays

Can also give names to the rows and columns of a matrix

```
rownames(x) <- c("A", "B", "C")
colnames(x) <- c("one", "two", "three", "four")
x
```

```
##   one two three four
## A   1   4     3     2
## B   2   1     4     3
## C   3   2     1     4
```

```
x[c("A"), c("one", "three")]
```

```
##   one three
##    1     3
```

```
x[c("A"), c("one", "three"), drop = FALSE]
```

```
##   one three
## A   1     3
```

```
x[-1, c("one", "three")]
```

```
##   one three
## B   2     4
## C   3     1
```

Note that `x[-c("A"), c("one", "three")]` will fail. Negation requires numerical indices (or logicals).

## Data structures in R – matrices and arrays

Can also give names to the dimensions of an array

```
dimnames(y) <- list(c("row 1", "row 2"),  
                    c("col 1", "col 2", "col 3"),  
                    c("slice 1", "slice 2"))
```

y

```
## , , slice 1  
##  
##      col 1 col 2 col 3  
## row 1    1    3    5  
## row 2    2    4    6  
##  
## , , slice 2  
##  
##      col 1 col 2 col 3  
## row 1    1    3    5  
## row 2    2    4    6
```

```
str(y)
```

```
## int [1:2, 1:3, 1:2] 1 2 3 4 5 6 1 2 3 4 ...  
## - attr(*, "dimnames")=List of 3  
## ..$ : chr [1:2] "row 1" "row 2"  
## ..$ : chr [1:3] "col 1" "col 2" "col 3"  
## ..$ : chr [1:2] "slice 1" "slice 2"
```

dimnames(x) will also work.

## Data structures in R – Data frames

A data frame is a list of equal-length vectors:

```
data <- data.frame(x = 1:3, y = 4:6, z=c("one", "two", "three"))
str(data)
```

```
## 'data.frame':    3 obs. of  3 variables:
## $ x: int  1 2 3
## $ y: int  4 5 6
## $ z: Factor w/ 3 levels "one","three",...: 1 3 2
```

Note that the strings are turned into factors. This can be suppressed:

```
data_nofactor <- data.frame(x = 1:3, y = 4:6, z=c("one", "two", "three"),
                           stringsAsFactors = FALSE)
str(data_nofactor)
```

```
## 'data.frame':    3 obs. of  3 variables:
## $ x: int  1 2 3
## $ y: int  4 5 6
## $ z: chr  "one" "two" "three"
```

A data frame can be thought of as a rectangular structure where each column is a variate and each row an observation. So it is similar to (but not the same as) a matrix.

## Data structures in R – Data frames

As a rectangular structure, a data frame behaves much like a matrix and, being a list, behaves much like one of those. Consider **selection** operators

```
data
```

```
##      x y      z  
## 1 1 4    one  
## 2 2 5    two  
## 3 3 6 three
```

```
data[1,]
```

```
##      x y      z  
## 1 1 4    one
```

```
data[, "x"]
```

```
## [1] 1 2 3
```

```
data$z
```

```
## [1] one  two  three  
## Levels: one three two
```

## Data structures in R – Data frames

As a rectangular structure, combining (and other matrix operators) should work.  
Effect of adding observations (rows).

```
data
```

```
##   x y   z  
## 1 1 4 one  
## 2 2 5 two  
## 3 3 6 three
```

```
moredata <- rbind(data, data[2,], c(1:(ncol(data)-1), "four"))
```

```
## Warning in `[<-factor`(`*tmp*`, ri, value = "four"): invalid factor level,  
## NA generated
```

```
moredata
```

```
##   x y   z  
## 1 1 4 one  
## 2 2 5 two  
## 3 3 6 three  
## 21 2 5 two  
## 5 1 2 <NA>
```

```
rownames(moredata)
```

```
## [1] "1" "2" "3" "21" "5"
```

Notes: 1. coercion and warning for the last row, and 2. fourth row name.

## Data structures in R – Data frames

Adding variates (columns).

```
data
```

```
##   x y   z  
## 1 1 4   one  
## 2 2 5   two  
## 3 3 6 three
```

```
moredata <- cbind(data, 100 * data, new = 1000)
```

```
## Warning in Ops.factor(left, right): '*' not meaningful for factors
```

```
moredata
```

```
##   x y   z   x   y   z   new  
## 1 1 4   one 100 400 NA 1000  
## 2 2 5   two 200 500 NA 1000  
## 3 3 6 three 300 600 NA 1000
```

```
colnames(moredata)
```

```
## [1] "x"   "y"   "z"   "x"   "y"   "z"   "new"
```

```
moredata$x
```

```
## [1] 1 2 3
```

Notes: 1. warning for multiplication, and 2. effect on column names.

## *Data structures in R – Data frames*

Data frames are vectors

```
data
```

```
##   x y   z  
## 1 1 4 one  
## 2 2 5 two  
## 3 3 6 three
```

```
attributes(data)
```

```
## $names  
## [1] "x" "y" "z"  
##  
## $class  
## [1] "data.frame"  
##  
## $row.names  
## [1] 1 2 3
```

```
length(data)
```

```
## [1] 3
```

```
typeof(data)
```

```
## [1] "list"
```

## Data structures in R – Data frames

Data frames are lists

```
data
```

```
##   x y   z  
## 1 1 4 one  
## 2 2 5 two  
## 3 3 6 three
```

```
class(data)
```

```
## [1] "data.frame"
```

```
is.data.frame(data)
```

```
## [1] TRUE
```

```
is.list(data)
```

```
## [1] TRUE
```

```
is.vector(data) # why?
```

```
## [1] FALSE
```



## *Data structures in R – Data frames*

Data frames are lists

```
data
```

```
##   x y   z  
## 1 1 4 one  
## 2 2 5 two  
## 3 3 6 three
```

```
data[1]
```

```
##   x  
## 1 1  
## 2 2  
## 3 3
```

```
data$x
```

```
## [1] 1 2 3
```

```
data[[1]]
```

```
## [1] 1 2 3
```

As a list, we might want to introduce some hierarchical structure.

## *Data structures in R – Data frames*

Provided they have the same length, each variate/component can be any atomic vector or list.

```
data_zAsList <- data.frame(x = 1:3, y = 4:6, z=list(1:3, 4:6, 7:9))
str(data_zAsList)
```

```
## 'data.frame':    3 obs. of  5 variables:
## $ x      : int  1 2 3
## $ y      : int  4 5 6
## $ z.1.3: int  1 2 3
## $ z.4.6: int  4 5 6
## $ z.7.9: int  7 8 9
```

But there is a problem.

The data frame has 5 instead of 3 variates.

The problem is that `data.frame()` interprets, as separate variates/components, each and every element of any component appearing as a list argument to `data.frame()`.

## *Data structures in R – Data frames*

The same thing can happen when the component/variable/column is a matrix or array.

```
data_zAsList <- data.frame(x = 1:3, y = 4:6, z = (matrix(1:12, nrow=3)))  
str(data_zAsList)
```

```
## 'data.frame':    3 obs. of  6 variables:  
## $ x : int  1 2 3  
## $ y : int  4 5 6  
## $ z.1: int  1 2 3  
## $ z.2: int  4 5 6  
## $ z.3: int  7 8 9  
## $ z.4: int 10 11 12
```

Again, the data frame has 6 instead of 3 variates.

## *Data structures in R – Data frames*

This can get you into trouble if elements of the list have different lengths.

```
data_zAsList <- data.frame(x = 1:3, y = 4:6, z=list(1:3, 4:5, 6:9))
```

```
Error in (function (... , row.names = NULL,
                      check.rows = FALSE,
                      check.names = TRUE,  :
arguments imply differing number of rows: 3, 2, 4
```

There are two solutions to this problem.

1. Use the “inhibit” function `I()` to stop `data.frame()` from processing the `z` list
2. Attach the `z` list to the `data.frame()` after it has been constructed.

## Data structures in R – Data frames

1. The “inhibit” function `I()` prepends the “AsIs” class to the object’s class. This stops its argument from being evaluated as its original class.

```
data_zAsList <- data.frame(x = 1:3, y = 4:6, z = I(list(1:3, 4:5, 6:9)))  
str(data_zAsList)
```

```
## 'data.frame':    3 obs. of  3 variables:  
## $ x: int  1 2 3  
## $ y: int  4 5 6  
## $ z:List of 3  
## ..$ : int  1 2 3  
## ..$ : int  4 5  
## ..$ : int  6 7 8 9  
## ..- attr(*, "class")= chr "AsIs"
```

## *Data structures in R – Data frames*

The effect when the component/variable/column is a matrix or array:

```
data_zAsList <- data.frame(x = 1:3, y = 4:6, z = I(matrix(1:12, nrow=3)))  
str(data_zAsList)
```

```
## 'data.frame':    3 obs. of  3 variables:  
## $ x: int  1 2 3  
## $ y: int  4 5 6  
## $ z: 'AsIs' int [1:3, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
```

## *Data structures in R – Data frames*

Or,

2. first create the data frame, and then add each list as a new component:

```
data_zAsList <- data.frame(x = 1:3, y = 4:6)
data_zAsList$z <- list(1:3, 4:5, 6:9)
str(data_zAsList)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ x: int  1 2 3
##  $ y: int  4 5 6
##  $ z:List of 3
##    ..$ : int  1 2 3
##    ..$ : int  4 5
##    ..$ : int  6 7 8 9
```

## Data structures in R – Data frames

This is more likely to arise when the rows/observations have more structure. For example, suppose each observation is a fitted model:

```
fit2cars <- data.frame(poly_degree = 1:3)
rownames(fit2cars) <- c("linear", "quadratic", "cubic")
fit2cars$fit_info <- lapply(fit2cars$poly_degree,
  FUN = function(p){
    fit <- lm(dist ~ poly(speed, p), data = cars)
    results <- summary(fit)$coefficients
    df <- data.frame(coefficients = results[, "Estimate"],
                     p_values = round(results[, "Pr(>|t|)"], 5)
    )
    df}
)
```

```
str(fit2cars)
```

```
## 'data.frame': 3 obs. of 2 variables:
## $ poly_degree: int 1 2 3
## $ fit_info :List of 3
## ..$ : 'data.frame': 2 obs. of 2 variables:
## .. ..$ coefficients: num 43 146
## .. ..$ p_values : num 0 0
## ..$ : 'data.frame': 3 obs. of 2 variables:
## .. ..$ coefficients: num 43 146 23
## .. ..$ p_values : num 0 0 0.136
## ..$ : 'data.frame': 4 obs. of 2 variables:
## .. ..$ coefficients: num 43 145.6 23 13.8
## .. ..$ p_values : num 0 0 0.137 0.369
```



## Data structures in R – Data frames

Then the value of the `fit_info` each observation is a fitted model

```
fit2cars
```

```
##           poly_degree
## linear           1
## quadratic        2
## cubic            3
##
##                                     fit_info
## linear                                42.9800, 145.5523, 0.0000, 0.0000
## quadratic                    42.98000, 145.55226, 22.99576, 0.00000, 0.00000, 0.13640
## cubic      42.98000, 145.55226, 22.99576, 13.79688, 0.00000, 0.00000, 0.13727, 0.36892
```

```
fit2cars$fit_info
```

```
## [[1]]
##           coefficients p_values
## (Intercept)      42.9800      0
## poly(speed, p)   145.5523      0
##
## [[2]]
##           coefficients p_values
## (Intercept)      42.98000  0.0000
## poly(speed, p)1   145.55226  0.0000
## poly(speed, p)2    22.99576  0.1364
##
## [[3]]
##           coefficients p_values
## (Intercept)      42.98000  0.00000
## poly(speed, p)1   145.55226  0.00000
## poly(speed, p)2    22.99576  0.13727
## poly(speed, p)3    13.79688  0.36892
```

## *Examining the data structure*

There are a few ways which you might use to examine the data structure.

## Examining the data structure

There are a few ways which you might use to examine the data structure.

Two we have already seen.

1. the printed representation which appears in the console (via `print()`) as

```
x <- 1:10  
x
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
```

```
# or equivalently as  
print(x)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
```

2. or if we want to see the structure of the object, we use `str()`

```
str(x)
```

```
## int [1:10] 1 2 3 4 5 6 7 8 9 10
```

which reveals the detailed structure of the object. This is particularly important (as we have already seen) for more complex objects such as those constructed using `list()`.

## Examining the data structure - `print()`

Note that `print()` is a **generic** function and is typically specialized according to the `class()` of the data structure.

```
# vector
print(seq(from = 2, to = 50, by = 10))
```

```
## [1] 2 12 22 32 42
```

```
# matrix
x <- matrix(rnorm(1000), nrow = 200)
print(head(x))
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,]  0.3088521 -1.013707  0.3869193 -1.4895012 -1.3099636
## [2,]  0.5378807 -1.399676 -0.6360947 -1.6821909  1.8251058
## [3,] -0.4611291 -1.375043 -0.2410696  0.6945442 -0.9068607
## [4,] -0.2496483 -1.142924  0.5998035  0.4235241  0.2790625
## [5,] -0.3999582 -1.481012 -0.7976473 -1.0203408 -0.7231094
## [6,]  2.0965151  1.155833 -0.8521571 -1.4092708  0.4136341
```

```
# array
x <- array(rnorm(1000), dim = c(20,5,10))
print(head(x))
```

```
## [1] 1.70249924 0.12435219 0.03221932 2.21592610 -0.26641531 -0.79918816
```

In each case, try just `print(x)` (or just `x`) (There is also a `tail()` function.)

## Examining the data structure - `print()`

```
# data frame
print(head(mtcars))
```

```
##           mpg  cyl  disp  hp  drat    wt   qsec  vs  am  gear  carb
## Mazda RX4      21.0   6  160 110  3.90  2.620 16.46  0   1    4    4
## Mazda RX4 Wag   21.0   6  160 110  3.90  2.875 17.02  0   1    4    4
## Datsun 710      22.8   4  108  93  3.85  2.320 18.61  1   1    4    1
## Hornet 4 Drive  21.4   6  258 110  3.08  3.215 19.44  1   0    3    1
## Hornet Sportabout 18.7   8  360 175  3.15  3.440 17.02  0   0    3    2
## Valiant         18.1   6  225 105  2.76  3.460 20.22  1   0    3    1
```

```
# a list
print(head(list(A = 1:3, b = LETTERS[1:4],
                c = list(d = letters[10:12], e = month.abb[1:3]))))
```

```
## $A
## [1] 1 2 3
##
## $b
## [1] "A" "B" "C" "D"
##
## $c
## $c$d
## [1] "j" "k" "l"
##
## $c$e
## [1] "Jan" "Feb" "Mar"
```

## Examining the data structure - *print()*

```
# A fitted model
x <- lm(mpg ~ wt + poly(displacement,3), data = mtcars)
print(x)

##
## Call:
## lm(formula = mpg ~ wt + poly(displacement, 3), data = mtcars)
##
## Coefficients:
##      (Intercept)              wt  poly(displacement, 3)1  poly(displacement, 3)2
##           24.250           -1.293           -22.186              9.211
##  poly(displacement, 3)3
##           -7.422
```

Exercise: What happens when you call `print.default(x)` in this case?

Note: `print()` prints its value and returns it “invisibly” (i.e. doesn’t return it unless it is actually assigned – see `?invisible`).

## *S3 classes, print(), and class names*

The `print()` function is implemented via numerous **S3 methods** such as `print.default()`, `print.data.frame()`, and `print.lm()` (the last is from the `stats` package) which are specialized by the `class()` of the first argument matching the function name after the first dot ..

Structures in R have an associated **S3 class** accessible via `class()`. This is the most prevalent (and basic) kind of class in R

Try the following:

```
# A fitted model
x <- lm(mpg ~ wt + poly(displacement,3), data = mtcars)
class(mtcars)
class(mtcars$mpg)
class(x)
class(lm)
```

Generic functions (like `print()`) can be implemented to dispatch on the class of some argument (typically the first argument) via methods which are functions of the same name appended by a “dot” and then class name.

## *S3 classes, print(), and class names*

For example, we can introduce the generic function `foo()` as follows

```
# A generic function
foo <- function(x, y) {
  UseMethod("foo", x)
}

foo.default <- function(x, y){
  x
}

foo.data.frame <- function(x, y) {
  y
}
foo(3, 4)
foo(mtcars, 4)
foo(3, mtcars)
```

Note: Could have dispatched on `y` instead (though this is not typical in R)



## *Examining the data structure - RStudio's View()*

In RStudio there is also a very useful function called `View()` which can be used to examine the value of any structure. For example, try

```
# A fitted model
x <- lm(mpg ~ wt + poly(displacement,3), data = mtcars)
View(mtcars)
View(mtcars$mpg)
View(x)
View(lm)
```

## Examining the data structure - summary()

It is often convenient to get a quick summary of data structure. To this end, R provides the **S3 generic function** `summary()`.

```
# A data frame  
summary(mtcars)
```

```
##      mpg          cyl          disp          hp  
## Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
## 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
## Median :19.20   Median :6.000   Median :196.3   Median :123.0  
## Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
## 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
## Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##      drat          wt          qsec          vs  
## Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
## 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
## Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
## Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
## 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
## Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##      am          gear          carb  
## Min.   :0.0000   Min.   :3.000   Min.   :1.000  
## 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
## Median :0.0000   Median :4.000   Median :2.000  
## Mean   :0.4062   Mean   :3.688   Mean   :2.812  
## 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
## Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

Exercise: What happens when you call `summary.default(mtcars)` in this case?

## Examining the data structure - summary()

Being a generic function, `summary()` can be specialized to provide informative summaries appropriate to the class of the object.

```
# A fitted model
x <- lm(mpg ~ wt + poly(displ,3), data = mtcars)
summary(x)

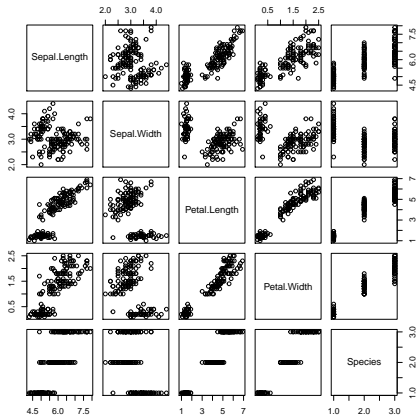
##
## Call:
## lm(formula = mpg ~ wt + poly(displ, 3), data = mtcars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.9946 -1.4464 -0.3745  1.4810  4.4319
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)      24.250      4.115   5.894  2.8e-06 ***
## wt              -1.293      1.273  -1.016  0.318836
## poly(displ, 3)1  -22.186      6.547  -3.389  0.002173 **
## poly(displ, 3)2   9.211      2.223   4.143  0.000303 ***
## poly(displ, 3)3  -7.422      3.189  -2.327  0.027694 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.222 on 27 degrees of freedom
## Multiple R-squared:  0.8816, Adjusted R-squared:  0.864
## F-statistic: 50.25 on 4 and 27 DF, p-value: 3.997e-12
```

Exercise: What happens when you call `summary.default(x)` in this case?

## Examining the data structure - `plot()`

Finally, a visual presentation of the data can be convenient and routine. To this end, R provides the **S3 generic function** `plot()`.

```
# A data frame  
plot(iris)
```

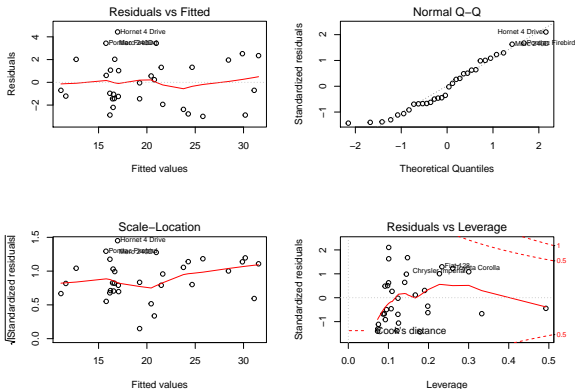


Exercise: What happens when you call `plot.default(iris)` in this case?

## Examining the data structure - `plot()`

As with other **S3** generic function `plot()` is specialized for different structures

```
# A fitted model
x <- lm(mpg ~ wt + poly(displ,3), data = mtcars)
par(mfrow = c(2,2)) # This will make more sense later
plot(x)
```



Exercise: What happens when you call `plot.default(x)` in this case? What does this say about default plotting?